Methodology for Splitting Business Capabilities into a Microservice Architecture: Design and Maintenance Using a Domain-Driven Approach

Benjamin Hippchen, Michael Schneider, Iris Landerer, Pascal Giessler
Sebastian Abeck

Cooperation & Management (C&M), Institute for Telematics
Karlsruhe Institute of Technology
Karlsruhe, Germany
{benjamin.hippchen, michael.schneider, iris.landerer9, pascal.giessler, abeck}@kit.edu

***Abstract*—The ongoing digital transformation is forcing organizations to rethink not only their business domains but also their (often monolithic) application landscapes. A more flexible architecture is needed: microservice architecture. Migrating, developing and operating such a flexible architecture requires predetermined architectural decisions. Because splitting the business domain into a more distributed software architecture is challenging, a methodology must be created that supports software architects by designing and systematically maintaining this kind of architecture. During our research, we discovered that there are only a few publications in this field that ignore the business domain and omit the maintenance of the architecture. Therefore, we provide a methodology for splitting business capabilities into a microservice architecture based on concepts of domain-driven design, which was proved over a longer time and continuously incorporated with new results. Our results indicate that we established a systematic and comprehensible creation process for microservice architecture, which also has a verifiable positive effect on the organization's application landscape.**

Keywords–Microservice; Microservice Architecture; Domain-Driven Design; Context Map; Bounded Context.

I. Introduction

The digital transformation is in progress and organizations must participate; otherwise, they will be left behind. Existing business models need to be rethought and new ones created. Tightly coupled to the business model is the organization's application landscape. Thus, this landscape also has to be reimagined. Meanwhile, microservice architectures have established themselves as an important architectural style and can be considered enablers of the digital transformation [1]. Therefore, one major step towards a digital organization is the migration of legacy applications into a microservice architecture. Afterwards, the architecture must be maintained to provide long-lived software systems. However, neither the migration, design and development of a microservice architecture nor its maintenance are easy to achieve.

The structure of the new microservice-based application seems straightforward for the development team. Some microservices communicate with each other and deliver business-related functionalities over web application interfaces (web APIs). However, at this point, the corresponding development team must ask itself decisive questions: How many microservices do we need? In which microservice do we put which functionality? Do we interact with third party applications? Domain-driven design (DDD) by Evans [2] can provides important concepts which help answering this questions. As a software engineering approach, DDD focuses on the customer's domain and wants to reflect this structure into the intended application. The business and its business objects are the focus of each developing activity. Technical details, like the deployment environment or technology decisions, are omitted and do not appear in design artifacts. Domain-driven design emphasizes the use of a domain model as a main development artifact: all relevant information about the domain, or business, is stored in it.

For microservice architecture, DDD helps structuring the application along business boundaries. Likely, these boundaries match the customer's domain boundaries. In his book *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Evans introduces the "context map" diagram. This diagram's main purpose is to explore the customer's domain and state it as visual elements. The context map focuses on the macro structure of the domain, sub domains, departments and so on instead of micro elements like business objects. A further essential DDD element and pattern is the "bounded context," which represents a container for domain information. This container is filled with the mentioned domain's micro structure, creating a domain model. The name bounded context is derived from its explicit boundary. Through this boundary, the container's content is only valid inside of the bounded context. From the strategic point of view, a bounded context is a candidate for a microservice. Thus, the context map could display the organisation's microservice architecture.

Like most DDD concepts, creating a context map is challenging and the tasks are not straightforward. The vague definitions and lack of process description create problems. The following example illustrates the problem. A development team wants to establish a microservice architecture at Karlsruhe Institute of Technology (KIT) for the administration of students. Typically, for this purpose, universities introduce Student Information Systems (SIS) to support the business process execution for their employees. There are several problems with those SIS: (1) in the hands of software companies, (2) little to no understanding of the university's domain, (3) primarily monolithic architecture, and, (4) little to no insights for third parties. Because the development team has no affect on the SIS and its architecture, the goal is to advance the SIS with social media aspects to support interaction between students. A microservice architecture is planned for the new functionality. Starting with the development and using DDD, the team must gather information about the domain and create a domain model and a context map. The first uncertainty is the artifacts creation order. Both artifacts rely on information from each other. While creating the domain model, the development team needs to know where to look for specific domain information, which is stated in the context map. When creating the context map, several bounded contexts are needed, which contain a domain model. In addition to this problem, the content of a context map is not precisely defined. The literature states that the context map contains bounded contexts and relationships but does not state how to elicit them or even what they represent in the real world. This lack of real-world representation is especially a problem for development teams, who need to interact with an existing application. On the one hand, it is necessary to provide the third party application in the context map, because the context map can capture the information transferred between the third-party and the university. On the other hand, it is unclear how to represent the third-party application in the context map. A bounded context needs a domain model, and there is no domain model in this case. These are only two problems with the application of the context map, but they illustrate how import it is to enhance usability. In the following sections, we discuss these and other problems in more detail.

In this paper, we provide the following contributions to enhance the application of the context map and support the design and maintenance of a microservice-based application:

- **Context Map Foundations**: One major problem of DDD is the lack of integration and placement in existing software development processes. It is unclear in which phases the context map must be created and in which phases it supports the development. Therefore, in Section III, we provide the first integration and placement of this map. In addition, we discuss the foundations of the context map and define the elements in this section.
- **Context Choreography**: While applying DDD for the development of microservice-based applications, we realized the existing artifacts did not capture all relevant information. Thus, in Section II and Section III, we introduce a new type of diagram, the "context choreography". This diagram's purpose is to display the choreography between multiple microservices for the application.
- **Artifact Creating Order**: As mentioned, it is unclear in which order the DDD artifacts must be created. Therefore, in Section III, we also provide a detailed order with an emphasis on the context map. The application of the order is presented in our case study in Section IV.

II. PLACEMENT AND INTEGRATION OF THE CONTEXT MAP

One main problem of DDD is its lack of placement in the field of software development. Neither its models nor its patterns, including the context map, are placed in common software development processes. For our placement, we focus on the context of microservices. Because the context map has some weakness in development, a new diagram is introduced to close the gap.

A. Placement

As mentioned in Section I, the use of the context map is not straightforward. The development team must analyze the domain, create a domain model, and develop a context map. On the one hand, the context map has a great benefit for microservice architectures. On the other hand, applying the map correctly is difficult.

Each DDD practice should be performed with the focus on an intended application [2]. This ensures the "perfect fit" of the gathered information, called "domain knowledge," for the application. Domain knowledge is captured in domain models. At this point, the pattern "bounded context" becomes important. An application consists of multiple bounded contexts, which all have their own domain models. With respect to the complexity of the domain knowledge, it makes sense to split the domain knowledge into multiple domain models. The validity of each domain model is limited through the bounded context. Furthermore, each bounded context has its own "ubiquitous language," which is based on the domain knowledge and acts as a contract for communication between project members and stakeholders. For the development of microservice-based applications especially, the multiple bounded contexts support the idea of a microservice architecture. Through connections between the bounded contexts, the domain knowledge is joined together in the application. The arising relationships are application-specific and differ from application to application. There are several types of relationships [3]. Modeling the bounded contexts and their relationships is the purpose of a context map.

Considering a microservice architecture, the purpose of a context map is not only to elicit domain knowledge. Organizations that introduce microsevices need to manage their application landscape to maintain the microservice architecture. Without the knowledge about which microservices are available and who is in charge of them, the microservice architecture loses its advantages. Existing microservices are simply not used, even the domain knowledge they provide is required, due to the fact that other development teams could not find it, oversee it or forgot about it. The required domain knowledge is redeveloped in new microservices and the existing microservices become legacy. Sustaining the advantages of a microservice architecture is therefore important for organizations and the context map is one tool which helps to achieve this. In addition, the aspect that DDD focus its development artifacts on the customer's domain, supports the maintenance of the microservice architecture. Aligning the context map to the customer's domain leads to a natural-looking architecture [4]. Conway's Law [5] also supports the idea behind a natural-looking microservice architecture. The organizational structure is adapted in the microservice architecture and vice versa. Looking at concepts like Martin Fowler's "HumaneRegistry" [6] or API management products like "apigee" [7], the idea and approach of the context map is required and furthermore it supports such concepts and products.

Using the context map as a tool for maintaining the microservice architecture is contrary to one DDD aspect: focus always on an application. The mentioned maintenance does not require any kind of application-specific information. A microservice is firstly an application-independent software building block [8] and needs to be treated as such while maintaining the microservice architecture. Even if the development of a microservice is motivated through the development of an application. Thus, we see the context map as an application-independent diagram.

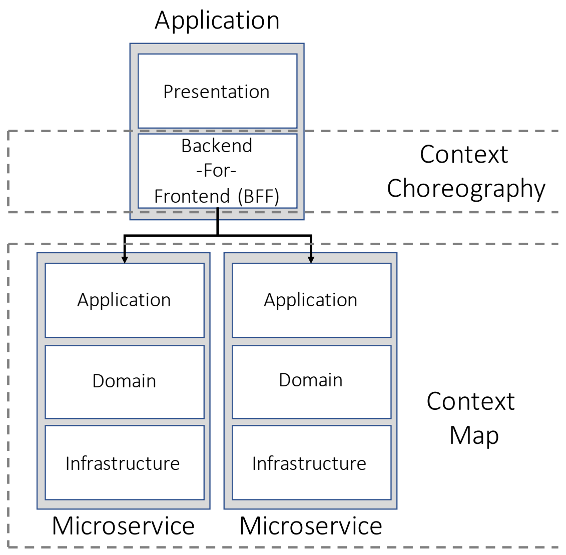

Figure 1. Placement of context mapping artifacts regarding the software building blocks from [8]

According to [8], for microservice-based applications, microservices are choreographed in applications through a backend-for-frontend (BFF) pattern. This is where application-specific information comes into play. Fig. 1 depicts the software architecture, including the application's BFF. To capture the choreography in the BFF, a new type of diagram is needed. The "context choreography" provides a view of the bounded context necessary for the application. Furthermore, the context choreography indicates which domain knowledge the bounded context transfers.

The context map can also be placed into software development activities. In [8], the first steps to place DDD into the software development activities from Brügge et al. [9] were taken. However, the context map itself was omitted. We built on these results for our placement of the context map. Domain-driven design introduces two types of "design activities" [2]. The first is the "strategic design," with tasks in modeling and structuring the domain's macroarchitecture (e.g., departments are used to define boundaries). This macroarchitecture is captured in the context map. Secondly, the "tactical design" further refines the macroarchitecture and enriches the bounded contexts with domain knowledge. This activity represents the microarchitecture of the domain and therefore of the microservice. Both activities rely on creating diagrams. Considering the software development activities from Brügge et al., Evans' designation as strategic and tactical "design" is misleading. Those focus more on the analysis than on the design phase. Many DDD practices and principles, such as "knowledge crunching," aim to analyze the domain. The development team explores the customer's domain and should simultaneously create the context map and domain model. Thus, the strategic and tactical designs are completed out, which is why the context map must be integrated at this point.

As mentioned, the content of the context map depends on its purpose. This is even the case for the relationships between the bounded contexts. Developing a monolithic application requires a different viewpoint on these relationships than a microservice-based application requires. A microservice architecture has many different microservices, which are managed by different development teams. By choreographing microservices in applications, development teams are automatically interdependent. This dependency is illustrated in the relationships in the context map. They could also be seen as communication paths between those development teams.

Our placement indicates that the context map has several possibilities to support the development of microservice architectures and microservice-based applications. We distinguish between a microservice architecture and the development of a microservice-based application. With regard to the microservice architecture style, the context map provides an overview of all in the application landscape existing microservices and further the dependencies of the responsible development teams—which are also necessary information for maintaining the microservice architecture. Due to the application-independence of these information, the context map is an application-independent diagram. Additionally, we saw a lack of the context map while specifying microservice-based applications. Information transferred between microservices was missing a specification, which is necessary for choreographing the BFF of the application. Thus, we introduced the context choreography, which displays the application-specific dependencies between the microservices and their transferred domain knowledge. With this placement, we make a first step in advancing the use of the context map.

III. Foundations and Artifact Order

In addition to the placement, we see a high need for clear definitions and guidance in creating the context choreography and context map. Therefore, this section provides definitions for terms regarding both artifacts. Afterward, we explain how the artifacts could be created.

A. Foundations

We found that, in addition to the development process, terminology around the context map is not clearly defined. This also leads to difficult application. Therefore, we want to provide some basics.

1) Bounded Context: The bounded context is the main element for the context map and is an explicit boundary for limiting the validity of domain knowledge [2]. Thus, within this context, there is a domain model and its ubiquitous language. A bounded context does not represent an application. This is based on the layered architecture of DDD, which consists of four layers: (1) presentation, (2) application, (3) domain, and (4) infrastructure. Domain-driven design and its artifacts focus only on the domain layer and omit the others. Therefore, without any application logic in a domain model, a bounded context cannot represent an application. This definition is necessary, when creating a context map. An intended application is usually integrated into an existing application landscape.

When developing a microservice-based application, a bounded context initially only represents a candidate for a microservice [4]. Thus, a bounded context is either large enough that two or more microservices are necessary or small enough that they are included in one microservice. The best practice, however, should be the one-to-one relationship. This relationship eases the maintenance of the architecture through a clearer mapping between bounded contexts, microservices, and the responsible development teams. Reconsidering the size of the bounded context helps achieve this mapping. Therefore, we have collected several indicators, or more precisely possible influence factors, for the size of bounded contexts from our experiences in research and practice. This list should not be considered complete or verified with an empirical study but should rather be seen as an aid. A bounded context (1) has a high cohesion and low coupling, (2) can be managed by one development team, (3) has ideally a high autonomy to reduce the communication/coordination effort between development teams, (4) has a unique language that is not (necessarily) shared, and (5) represents a meaningful excerpt of the domain.

2) Context choreography: As mentioned (see Section II), the specification of a microservice-based application was lacking some information. Thus, we introduced the context choreography as a new diagram.

For microservice-based application development, it is important to state the other needed microservices—and thus the bounded context also. Furthermore, the exchanged data between those microservices are important information. As a microservice-based application is developed, existing microservices could still be used, while new ones are developed. In both cases, the context choreography is supportive. Regarding the application itself, the context choreography states all necessary microservices and the transferred domain knowledge between them, literally displaying the choreography of the microservices to achieve the application functionality. According to the software architecture provided by [8], the application's BFF is specified. Independent from the application, the context choreography states the microservice interfaces. Both the consumed and the provided interfaces of the microservice are provided. Thus, while developing the application, the first hints of the API can be derived. With regard to the subsequent maintenance of the microservice, development teams are able to identify the microservices that rely on them and vice versa.

3) Context Map: The DDD's original purpose for the context map differs from the one provided in this paper. In the context of microservice architecture, the context map is a useful diagram for maintaining the architecture and supporting application development.

One major advantage is the comprehensive overview of existing microservices. According to the best practice from Section III-A1, each bounded context in the context map represents a microservice. Further, in software architecture, social and organization aspects have to be considered [10]. Therefore, dependencies between microservices, and thus development teams, are stated. When development teams want to evolve their microservices, it is important to ask who depends on these microservices. At this point, the dependencies on other teams must be considered because any change could affect the stability of the other microservices. Thus, changes have to be communicated.

Also, for the development of a microservice-based application, the context map is advantageous. Regarding the context choreography, existing microservices are used to compose functionality for the intended application. Using existing microservices is only possible if they are traceable in the microservice architecture. This is where the context map comes into play. After developing a microservice, it is placed as a bounded context into the model. While the application is in development, the development team can use the context map as a tool to locate the needed microservices.

4) Domain Experts and other Target Groups: The interaction between domain experts and developers is one principle of DDD [2]. Each artifact is created for and with domain experts. Thus, the artifacts should be understandable without a software development background.

The context map according to DDD's definition is also relevant for domain experts [11]. However, according to our definition, we do not see any advantages for domain experts since the context map provides an overview of bounded contexts and communication paths between development teams. Furthermore, the context choreography is irrelevant. Only the subdomains, which represent the organization's structure, contain helpful information.

B. Process for Establishing a Context Map

To develop a microservice-based application, it is necessary to establish the bounded contexts needed for the application. The developed application also may reuse existing microservices, which should be integrated into the application landscape. To obtain an overview of the microservice landscape, the context map is useful. In this section, we focus on the establishment of the bounded contexts, the context choreography, and the context map. For developing an application, we build on a development process based on behavior-driven development (BDD) [12] and DDD [2] introduced in [8]. We omit the steps in [8] and focus on the creation of context choreography and a context map. Therefore, this section describes how the context map is established and enhanced within the development process.

1) Forming the Initial Domain Model: Forming the initial domain model occurs in the analysis and design phases. Before developing an application, the requirements are specified with BDD in the form of features. As Fig. 2 illustrates, a tactical diagram is derived from the features (e.g., the domain objects and their relationships). If a domain model already exists (e.g., from an existing microservice), this should be taken into account. The resulting diagram represents the initial domain model, which contains the application's business logic. Thus, the domain model provides the semantic foundation for all the specified features. The resulting diagram is comparable to a Unified Modeling Language (UML) class diagram and displays the structural aspects of the domain objects. If the domain structure is still vague when the number of features is satisfied, more features are considered until the domain model appears to be meaningful. Afterward, as presented in Fig. 2, this initial domain model is examined and structured into several bounded contexts.

2) Forming the Bounded Contexts: The model is strategical analyzed and separated based on the business and its functionality. This step depends on the domain knowledge and the structure of the business. Therefore, knowledge crunching from DDD [2] is applied to gather that knowledge. Often, a business's domain knowledge is scattered through the whole business. Therefore, analyzing the business is important to

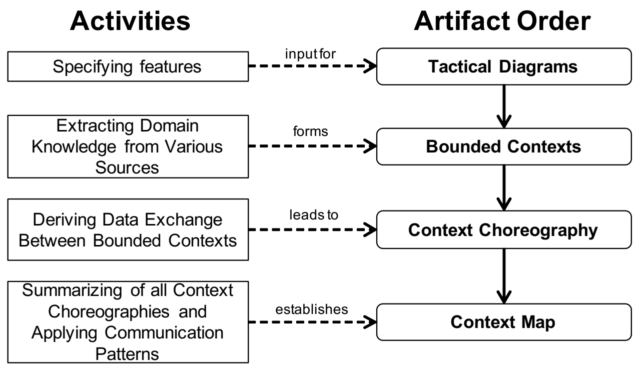
Activities
Artifact Order
Specifying features
input for
Tactical Diagrams
Extracting Domain Knowledge from Various Sources
forms
Bounded Contexts
Deriving Data Exchange Between Bounded Contexts
leads to
Context Choreography
Summarizing of all Context Choreographies and Applying Communication Patterns
establishes
Context Map

Figure 2. Creating order for artifacts their and impacting activities

understand the business processes and the interaction of different departments. By default, each department knows its tasks the best. To extract the domain knowledge, various sources should be considered. These sources include domain experts who are part of a department, as well as documents and organizational aspects. This domain knowledge provides hints for structuring the domain and has to be considered while forming the bounded contexts. Considering the application analysis and the business analysis from [8] leads to the bounded contexts, as illustrated in Figure 2. If a context map has been established, then the context map is searched for the required domain knowledge of the application. If a bounded context representing the domain knowledge already exists, then this bounded context is taken into account. A new bounded context is established if the context map does not contain the required domain knowledge. For example, we integrated a profile context into an existing context map of the campus management domain.

3) Toward the Context Choreography: Forming the bounded contexts is only the first step towards a working application. Each previously established bounded context is considered a microservice and requires or offers a unique interface for communication that can be based on REST or other architectural styles. Without interfaces, a microservice-based application would not work. The microservices are choreographed with the BFF. To allow choreography, the data exchange between the bounded contexts is considered next. The required data exchange is modeled in the context choreography. For each bounded context, a context choreography diagram is modeled. Domain objects that need to be shared or consumed from other bounded contexts are modeled as shared entities. The considered bounded context can either share the domain object itself or consume the domain object. This model also provides hints for the API of a bounded context if the bounded context shares entities.

4) Toward the Context Map: In microservice architectures, each established bounded context represents a microservice and is implemented by autonomous development teams. Thus, for relating bounded contexts, teams may need to communicate with each other. Therefore, the communication effort between the teams should be considered. The communication effort indicates how much communication between the teams is required. Clear communication paths are necessary, because a team needs to know which other team is responsible for relating bounded contexts. Therefore, dependencies and communication channels between teams are defined. Depending on the teams and the possible communication effort, a communication pattern is chosen based on [2] [3] (see Table I). The last three communication patterns listed in Table I are special patterns designed to reduce the communication between different teams, as well as the impact on interface changes. Other benefits and drawbacks of particular patterns exist but they are out of the scope for the current discussion. The context map illustrates the determined communication path between the bounded contexts. For example, when the communication between teams is not possible, such as when foreign services are adopted, DDD patterns need to be applied. For foreign services, the ACL pattern should be applied. In the last step, as depicted in Figure 2, the relationships (including the pattern) and the bounded contexts are added to the context map diagram.

TABLE I. Overview of communication patterns and their impact

Comm. pattern	Description	Effort
Partnership	Cooperation between bounded contexts to avoid failure	Very high
Shared Kernel	Explicit shared functionality between different development teams	Medium to high
Customer/ Supplier	Supplier provides required functionality for the customer. The customer has influence on the supplier's design decisions	High to very high
Conformist	Similar to customer/supplier but with no influence on design decisions.	Low to medium
Separate Ways	No cooperation between development teams	Low
Anticorruption Layer	Additional layer that transforms one context into another	Low
Open Host Service	Uniform interface for accessing the bounded context	Low
Published Language	Information exchange is achieved using the ubiquitous language of the bounded context	Low

Adding those bounded contexts and communication relationships is an essential part of the context map. This concludes the first cycle of the analysis and design phases.

5) Adjustments of the models: After the design phase, the implementation phase follows. In this phase, the models are implemented and tested. Afterward, specific parts of the application are developed. Following the iterative process, new features are implemented into the next cycle. These features need to be analyzed and may change the domain model. In addition, this may lead to the establishment of new bounded contexts. Thus, the models, including the tactical diagrams, the context choreography, and the context map, are refined according to the features and the knowledge crunching process in the previous steps.

IV. Case Study: Campus Management

In our case study, we illustrate our approach of establishing a new bounded context and integrating it into an existing context map. The case study orientates itself on the process as described in Section III. Over three semesters, we expanded the campus management system of KIT with microservice-based applications. The case study represents our recent project in this field and adds a social media component to the campus management system.

A. Project Scope

Our vision is to simply and efficiently support the exchange of information and facilitate cooperation between students. For this purpose, we wanted to introduce a profile service in the campus management system. This profile service should allow students to create custom profiles to display information about their studies, like currently visited lectures and future exams. The purpose of this service is to assist students with their studies and their search for learning partners. For example, students can find learning partners with the help of the profile service when other students share the lectures they attend.

B. Requirement Elicitation

In our process, we began by eliciting the requirements with BDD and formulating them as features. Fig. 3 presents one of the main features that enables students to edit their profile.

1. **Feature**: Providing student profiles
2. As a student
3. I can provide relevant information about myself
4. So that others can see my interests and study information

5. **Scenario**: Publish profile
6. **Given** I was never logged in to the ProfileService
7. **When** I log in to the ProfileService for the first time
8. **Then** my study account is linked to the ProfileService
9. **And** I choose which profile information I want to publish

Figure 3. Example of a BDD feature for publishing an user profile

C. Initial Domain Model

Analyzing the features leads to the initial domain model by deriving domain objects and their relationships. In our previously defined feature (see Section IV-B), we identified, the terms "Profile," "Examination," and "Student" and added them to the initial domain model. By repeating this procedure with all features, the domain model is enriched with the business logic. The result of the initial tactical diagram is presented in Fig. 4.

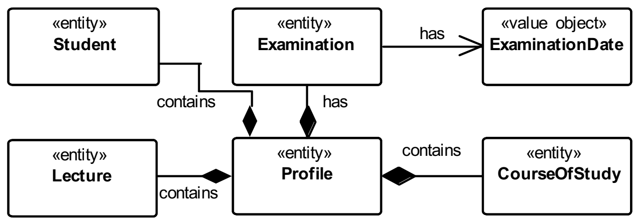

Figure 4. Initial domain model derived from BDD features and other sources

D. Bounded Contexts and Context Choreography

While we analyzed the domain, we also considered the existing context map of the campus management domain. We noticed that the bounded contexts "StudentManagement," "ExaminationManagement," "ModuleManagement," and "CourseMapping" already offered the required functionality. Only "ProfileManagement" had to be established as a new bounded context. Therefore, we considered the data exchange between the bounded contexts and created the context choreography on that basis. The result is illustrated in Fig. 5. The existing bounded contexts provide the required data as shared entities. The new bounded context "Profiles" adapts the shared

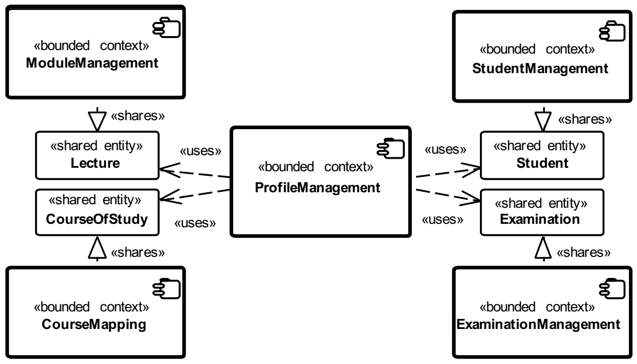
«bounded context» ModuleManagement
«bounded context» StudentManagement
«shares»
«shares»
«shared entity» Lecture
«uses»
«bounded context» ProfileManagement
«uses»
«shared entity» Student
«shared entity» CourseOfStudy
«uses»
«uses»
«shared entity» Examination
«shares»
«shares»
«bounded context» CourseMapping
«bounded context» ExaminationManagement

Figure 5. New bounded context "ProfileManagement" in a context choreography

entities and delivers the data required from the profile service. Last, the microservices are choreographed in the BFF of the intended application, to achieve the required application logic.

E. Integrating in Context Map

After we had established the bounded contexts and the context choreography, we needed to add the profile management context to the context map. Therefore, we determined the dependencies and communication channels between bounded contexts based on the context choreography. We found our development team did not influence any other bounded context. Thus, we applied the conformist as communication pattern. As a result, the context map depicted in Figure 6 was enhanced with the "Profiles" context. Afterward, we started the first

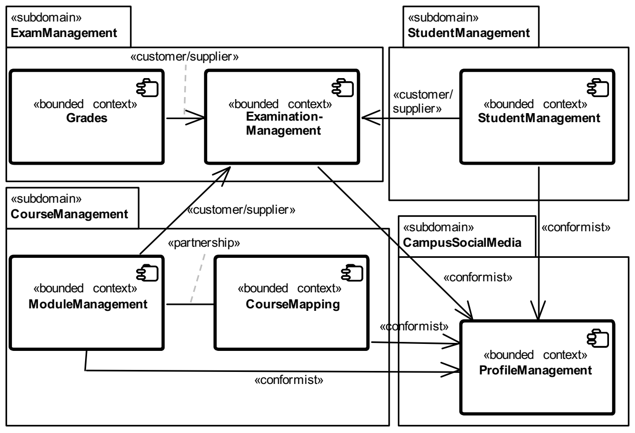
«subdomain» ExamManagement
«subdomain» StudentManagement
«customer/supplier»
«bounded context» Grades
«bounded context» Examination-Management
«customer/ supplier»
«bounded context» StudentManagement
«subdomain» CourseManagement
«customer/supplier»
«subdomain» CampusSocialMedia
«conformist»
«partnership»
«conformist»
«bounded context» ModuleManagement
«bounded context» CourseMapping
«conformist»
«bounded context» ProfileManagement
«conformist»

Figure 6. "ProfileManagement" context integrated into context map

implementation cycle.

F. Context Map as Template for a Deployment Map

The resulting context map provides an overview of the microservices that need to be deployed. Assuming that each bounded context represents one microservice, we can enhance the context map with technical information that is needed for the deployment in a secure manner. For instance, we can define which ports listen for incoming or are allowed for outgoing requests. By following such an approach each microservice is initially considered in isolation. We enforce this by defining default policies on the execution environment that need to be taken into account during deployment. The enhancement with technical information is transferred into a new diagram called a deployment map. For the modeling, we use a UML deployment diagram. In addition to a general overview of the deployment, it can also be used for an upfront security audit. We are planning to present the derivation rules in an upcoming publication.

For testing purpose, we have used a hosted Kubernetes [13] cluster on a cloud provider. Kubernetes is an open source system for provisioning and management of container-based applications and aroused from the collected experience behind Omega and Borg. First of all, we have defined policies to deny all ingress and egress traffic to all running Pods by default. A Pod groups one or more container and can be seen as the central brick of Kubernetes when deploying applications. Each bounded context will be represented by a microservice running in a container (Docker or rkt). Depending on their relation to each other (see TABLE I), we put them in corresponding Pods. Next, we have used the ports for incoming and outgoing traffic to derive the network policies. Finally, we have defined the services that wrap the Pods and offer a central access point for interaction. The application shows us that the underlying context map can be used as a basic scaffolding for deriving the deployment map but need to be enhanced with technical information as well as information from the development teams that realize the microservices. For instance, a persistent storage is missing in a context map due to its domain orientation but is needed for the deployment map.

V. Related Work

During our research, we searched for works comparable to the context map and its purpose. We encountered several inspiring works regarding different aspects of the context map. Especially, the focus on the microservice architecture is an important part of this paper.

A. Microservice Architecture

A microservice should concentrate on the fulfilment of one task and should be small, so a team of five to seven developers can be responsible for the microservice's implementation [4]. A microservice itself is not an application, but rather a software building block [8]. In microservice architecture, applications are realized through choreography of these building blocks. A central aspect of microservice architecture is the autonomy of the single microservices [14]. Each microservice is developed and released independently to achieve continuous integration.

B. Approaches for Designing a Microservice Architecture

The objective of microservice architectures is to subdivide large components into smaller ones to reduce complexity and create more clarity in the single elements of the system [14]. In this paper, we described our approach of designing microservice architecture with a context map from DDD. However, there are further strategies to identify microservices, which we considered in this paper.

One possible approach is event storming, as introduced by Alberto Brandolini in the context of DDD [15]. Event storming is a workshop-based group technique to quickly determine the domain of a software program. The group starts with the workshop by "storming out" all domain events. A domain event covers every topic of interest to a domain expert. Afterward, the group adds the commands that cause these events. Then, the group detects aggregates, which accept commands and accomplish events, and begins to cluster them together into bounded contexts. Finally, the relationships between bounded contexts are considered to create a context map. Like our approach, this strategy is based on DDD and results in a context map displaying the bounded contexts. Instead of a workshop for exploring the domain and defining domain events, we develop our bounded contexts through an iterative analysis and design phase. Furthermore, we enhanced the context map with maintenance aspect for microservice discovery and dependencies between development teams. The purpose of the resulting context map from [15] is comparable to the context choreography. Both focusing on the interactions between bounded context and identify the transferred domain knowledge.

Another method for approaching a microservice architecture is described in [16]. First, required system operations and state variables are identified based on use case specifications of software requirements. System operations define public operations, which comprise the system's API, and state variables contain information that system operations read or write. The relationships between these systems operations and state variables are detected and then are visualized as a graph. The visualization enables developers to build clusters of dense relationships, which are weakly connected to other clusters. Each cluster is considered a candidate for a microservice. This bears a resemblance to our approach because we also begin by focusing on the software requirements and take visualization for better understanding the domain.

A further widely used illustration of partitioning monolithic applications is a scaling cube, which uses a three-dimensional approach as described in [17]. Here, Y-axis scaling is important because it splits a monolithic application into a set of services. Each service implements a set of related functionalities. There are different ways to decompose the application which differ from our domain-driven approach. One approach is to use verb-based decomposition and define services that implement a single use case. The other possibility is to partition the application by nouns and establish services liable for all operations related to a specific entity. An application might use a combination of verb-based and noun-based decomposition. X-axis and Z-axis regards the operation of the microservices. The X-axis describes the horizontal scaling which means cloning and load balancing the same microservice into multiple instances. Meanwhile, the Z-axis denotes the degree of data separation. Both axis are important for microservice architectures and currently omitted in our context map approach.

C. Software Development Approaches

The development process we apply is based on BDD and DDD. As a method of agile software development, BDD should specify a software system by referencing its behavior. The basic artifact of BDD is the feature, which describes a functionality of the application. The use of natural language

and predefined keywords allows the developer to create features directly with the customer [18]. During our analysis phase, we used BDD for requirement elicitation.
In our design phase, we applied DDD based on the features we defined with BDD. DDD's main focus is the domain and the domain's functionality, rather than technical aspects [11]. The central design artifact is the domain model, which represents the target domain. In his book *Domain Driven Design - Tackling Complexity in the Heart of Software*, Eric Evans describes patterns, principles, and activities that support the process of software development [2]. Although DDD is not tied to a specific software development process, it is orientated toward agile software development. In particular, DDD requires iterative software development and a close collaboration between developers and domain experts.

D. Application of the Context Map

The goal of a context map, which Evans describes as one main activity of DDD, is to structure the target domain [2]. For this purpose, the domain is classified into subdomains, and in those subdomains, the boarders and interfaces of possible bounded contexts are defined. A bounded context is a candidate for a microservice, and one team is responsible for its development and operations [4]. Moreover, the relationships between bounded contexts are defined in a context map. Both the technical relationships and the organizational dependencies between different teams are considered.
A further aspect of the context map involves the maintenance of the miscroservice architecture. Without managing the application (or service) landscape, existing microservices are not used, even if they provide needed domain knowledge. The usage of a context map helps concepts like humane registry or API management products which tries to achieve maintenance goals. Martin Fowler introduced humane registry as a place, where both developers and users can record information about a particular service in a wiki [6]. In addition, some information can be collected automatically, e.g., by evaluating data from source code control and issue tracking systems. API management products like "apigee" [7] reach maintenance by pre-defining API guidelines such as key validation, quota management, transformation, authorization, and access control.

VI. Limitations and Conclusion

The concepts we provide in this in paper have some limitations. These are addressed in the next section. Afterward, we provide a short conclusion discussing our results.

A. Limitations

Domain-driven design has no special application or architectural style in mind. The concepts should be applied to DDD's layered architecture but could be applied to different architectural styles. For a better fit while developing microservice-based applications, we always had the microservice architecture in mind. Therefore, our provided concepts are only valid when developing a microservice-based application.

The concepts provided by this paper are built from our experiences which we gathered in various projects. Most of our projects were in the academic branch, but we also worked with industrial partners. For the context map, we developed and proved our concepts over a longer time. The case study describes our last project. Project members and partners gave us useful feedback about the concepts when they applied them. In addition, the feedback also included points we had not yet addressed, like a modeling language for context mapping. Nevertheless, evidence of our concepts in large microservice architectures, such as 50 or more microservices, still lacks. Our goal is to obtain prove for large microservice architectures in such projects.

Another limitation to our concepts is we only applied them in "clean" microservice architectures. However, in the real world, there are also legacy applications in the application landscapes of organizations. Typically, a legacy application is not a microservice-based one; often, it is a monolithic architecture. In future work, we must determine how to integrate legacy applications into the context choreography and context map.

B. Conclusion

During our research, we found many different studies that consider model-driven approaches for developing microservices. Using these approaches for microservices is common. In domain-driven design, especially, the approaches focus on the development itself but omit the design and maintenance phases. Therefore, we wanted to provide details on the design and maintenance of a microservice architecture using DDD's context map.

The context map has great potential to aid in developing and maintaining applications and is more useful when considering a microservice architecture. However, the context map shares a problem with most DDD concepts: its lacking placement in software engineering, foundations and concrete guidelines. Therefore, we first provided placement for the context map. Next, we clarified its foundations with a focus on the bounded context, the main concept of the context map. After the foundations were clear, we could develop a systematic approach for creating the context map. This approach began in the analysis phase with an initial domain model, separating the domain knowledge into bounded contexts, stating relationships between them, and putting the bounded contexts into a context map. The separation of bounded contexts and their relationships are stated in our new diagram, the "context choreography." This diagram's purpose is to illustrate necessary bounded contexts for microservice-based applications.

This paper's contributions are the first step in making the use of the context map, and now the context choreography, more efficient. Nevertheless, we see more opportunities for research, like a modeling language for the context map. Such a modeling language could be UML.

Acknowledgment

We want to give special thanks to Chris Irrgang and Tobias Hülsken for always providing their opinions and useful feedback on our concepts. Furthermore, we would like to thank the following development team, which provided the results in our case study: Alessa Radkohl, Nico Peter, and Stefan Throner.

References

[1] M. Gebhart, P. Giessler, and S. Abeck, "Challenges of the Digital Transformation in Software Engineering," *ICSEA 2016*, p. 149, 2016.

[2] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[3] V. Vernon, Ed., *Implementing Domain-Driven Design*. Addison-Wesley, 2013.

[4] S. Newman, *Building Microservices: Designing Fine-grained Systems*. " O'Reilly Media, Inc.", 2015.

[5] M. E. Conway, "How do Committees Invent," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[6] M. Fowler, "HumaneRegistry," URL: https://martinfowler.com/bliki/HumaneRegistry.html [retrieved: 02, 2019].

[7] Google, "apigee, API management," https://apigee.com/api-management/ [retrieved: 02, 2019].

[8] B. Hippchen, P. Giessler, R. Steinegger, M. Schneider, and S. Abeck, "Designing Microservice-Based Applications by Using a Domain-Driven Design Approach," in *International Journal on Advances in Software, Vol. 10, No. 3&4, pp. 432–445*, 2017.

[9] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns and Java-(Required)*. Prentice Hall, 2004.

[10] O. Vogel, I. Arnold, A. Chughtai, and T. Kehrer, *Software Architecture: A Comprehensive Framework and Guide for Practitioners*. Springer Science & Business Media, 2011.

[11] S. Millett, *Patterns, Principles and Practices of Domain-Driven Design*. John Wiley & Sons, 2015.

[12] J. F. Smart, *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning, 2015.

[13] K. Hightower, B. Burns, and J. Beda, *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*. O'Reilly Media, 2017.

[14] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. " O'Reilly Media, Inc.", 2016.

[15] A. Brandolini, "Introducing Event Storming," *blog, Ziobrando's Lair*, vol. 18, 2013, URL: http://ziobrando.blogspot.com/2013/11/introducing-event-storming.html [retrieved: 02, 2019].

[16] S. Tyszberowicz, R. Heinrich, B. Liu, and Z. Liu, "Identifying Microservices Using Functional Decomposition," pp. 50–65, 2018.

[17] N. Dmitry and S.-S. Manfred, "On Micro-Services Architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, 2014.

[18] M. Wynne, A. Hellesoy, and S. Tooke, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2017.